The InputStream Interface in C-Sharp

An `InputStream` is created using the following function:

C#

```
namespace Ice
{
    public sealed class Util
    {
        public static InputStream createInputStream(
            Communicator communicator,
            byte[] bytes);

        public static InputStream createInputStream(
            Communicator communicator,
            byte[] bytes,
            EncodingVersion version);

        public static InputStream wrapInputStream(
            Communicator communicator,
            byte[] bytes);

        public static InputStream wrapInputStream(
            Communicator communicator,
            byte[] bytes,
            EncodingVersion version);
    }
}
```

You can optionally specify an encoding version for the stream, otherwise the stream uses the communicator's default encoding version.

Note that the `createInputStream` functions make a copy of the supplied data, whereas the `wrapInputStream` functions avoid copies by keeping a reference to the data. If you use `wrapInputStream`, it is your responsibility to ensure that the memory buffer remains unmodified for the lifetime of the `InputStream` object.

The `InputStream` interface is shown below.

C#

```
namespace Ice
{
    public interface InputStream
    {
        Communicator communicator();

        void sliceObjects(bool slice);

        bool readBool();
        bool[] readBoolSeq();

        byte readByte();
        byte[] readByteSeq();

        short readShort();
        short[] readShortSeq();

        int readInt();
        int[] readIntSeq();

        long readLong();
        long[] readLongSeq();

        float readFloat();
```

```
        float[] readFloatSeq();

        double readDouble();
        double[] readDoubleSeq();

        string readString();
        string[] readStringSeq();

        int readSize();
        int readAndCheckSeqSize(int minSize);

        ObjectPrx readProxy();

        void readObject(ReadObjectCallback cb);

        int readEnum(int maxValue);

        void throwException();
        void throwException(UserExceptionReaderFactory factory);

        void startObject();
        SlicedData endObject(bool preserve);

        void startException();
        SlicedData endException(bool preserve);

        string startSlice();
        void endSlice();
        void skipSlice();

        EncodingVersion startEncapsulation();
        void endEncapsulation();
        EncodingVersion skipEncapsulation();
        EncodingVersion getEncoding();

        void readPendingObjects();

        object readSerializable();

        void rewind();

        void skip(int sz);
        void skipSize();

        bool readOptional(int tag, OptionalFormat format);

        int pos();

        void destroy();
    }
}
```

Member functions are provided for extracting all of the primitive types, as well as sequences of primitive types; these are self-explanatory. The remaining member functions have the following semantics:

- `void sliceObjects(boolean slice)`
  Determines the behavior of the stream when extracting Ice objects. An Ice object is "sliced" when a factory cannot be found for a Slice type ID, resulting in the creation of an object of a less-derived type. Slicing is typically disabled when the application expects all object factories to be present, in which case the exception `NoObjectFactoryException` is raised. The default behavior is to allow slicing.

- `int readSize()`
  The Ice encoding has a compact representation to indicate size. This function extracts a size and returns it as an integer.

- `int readAndCheckSeqSize(int minWireSize)`
  Like `readSize`, this function reads a size and returns it, but also verifies that there is enough data remaining in the unmarshaling buffer to successfully unmarshal the elements of the sequence. The `minWireSize` parameter indicates the smallest possible on-the-wire representation of a single sequence element. If the unmarshaling buffer contains insufficient data to unmarshal the sequence, the function throws `UnmarshalOutOfBoundsException`.

- `Ice.ObjectPrx readProxy()`
  This function returns an instance of the base proxy type, `ObjectPrx`. The Slice compiler optionally generates helper functions to extract proxies of user-defined types.

- `void readObject(ReadObjectCallback cb)`
  The Ice encoding for class instances requires extraction to occur in stages. The `readObject` function accepts a callback object of type `ReadObjectCallback`, whose definition is shown below:

  C#

  ```
  namespace Ice {
      public interface ReadObjectCallback     {
          void invoke(Ice.Object obj);
      }
  }
  ```

  When the object instance is available, the callback object's `invoke` member function is called. The application must call `readPendingObjects` to ensure that all instances are properly extracted. Note that applications rarely need to invoke this member function directly; the helper functions generated by the Slice compiler are easier to use.

- `int readEnum(int maxValue)`
  Unmarshals the integer value of an enumerator. The `maxValue` argument represents the highest enumerator value in the enumeration. Consider the following definitions:

  Slice

  ```
  enum Color { red, green, blue };
  enum Fruit { Apple, Pear=3, Orange };
  ```

  The maximum value for `Color` is 2, and the maximum value for `Fruit` is 4.

- `void throwException()`
  `void throwException(UserExceptionReaderFactory factory)`
  These functions extract a user exception from the stream and throw it. If the stored exception is of an unknown type, the functions attempt to extract and throw a less-derived exception. If that also fails, an exception is thrown: for the 1.0 encoding, the exception is `UnmarshalOutOfBoundsException`, for the 1.1 encoding, the exception is `UnknownUserException`.

- `void startObject()`
  `SlicedData endObject(bool preserve)`
  The `startObject` method must be called prior to reading the slices of an object. The `endObject` method must be called after all slices have been read. Pass true to `endObject` in order to preserve the slices of any unknown more-derived types, or false to discard the slices. If `preserve` is true and the stream actually preserved any slices, the return value of `endObject` is a non-nil `SlicedData` object that encapsulates the slice data. If the caller later wishes to forward the object with any preserved slices intact, it must supply this `SlicedData` object to the output stream.

- `void startException()`
  `SlicedData endException(bool preserve)`
  The `startException` method must be called prior to reading the slices of an exception. The `endException` method must be called after all slices have been read. Pass true to `endException` in order to preserve the slices of any unknown more-derived types, or false to discard the slices. If `preserve` is true and the stream actually preserved any slices, the return value of `endException` is a non-nil `SlicedData` object that encapsulates the slice data. If the caller later wishes to forward the exception with any preserved slices intact, it must supply this `SlicedData` object to the output stream.

- `string startSlice()`
  `void endSlice()`
  `void skipSlice()`
  Start, end, and skip a slice of member data, respectively. These functions are used when manually extracting the slices of an Ice object or user exception. The `startSlice` method returns the type ID of the next slice, which may be an empty string depending on the format used to encode the object or exception.

- `EncodingVersion startEncapsulation()`
  `void endEncapsulation()`
  `EncodingVersion skipEncapsulation()`
  Start, end, and skip an encapsulation, respectively. The `startEncapsulation` and `skipEncapsulation` methods return the encoding version used to encode the contents of the encapsulation.

- `EncodingVersion getEncoding()`
  Returns the encoding version currently in use by the stream.

- `void readPendingObjects()`
  An application must call this function after all other data has been extracted, but only if Ice objects were encoded. This function extracts the state of Ice objects and invokes their corresponding callback objects (see `readObject`). Note that `endEncapsulation` implicitly calls `readPendingObjects` if necessary.

- `void readPendingObjects()`
  An application must call this function after all other data has been extracted, but only if Ice objects were encoded. This function extracts the state of Ice objects and invokes their corresponding callback objects (see `readObject`). For backward compatibility with encoding version 1.0, this function must only be called when non-optional data members or parameters use class types.

- `object readSerializable()`
  Reads a serializable .NET object from the stream.

- `void rewind()`
  Resets the position of the stream to the beginning.

- `void skip(int sz)`
  Skips the given number of bytes.

- `void skipSize()`
  Reads a size at the current position and skips that number of bytes.

- `bool readOptional(int tag, OptionalFormat fmt)`
  Returns true if an optional value with the given tag and format is present, or false otherwise. If this method returns true, the data associated with that optional value must be read next. Optional values must be read in order by tag from least to greatest. The `OptionalFormat` enumeration is defined as follows:

  C#

  ```
  namespace Ice {
      enum OptionalFormat {
          OptionalFormatF1, OptionalFormatF2, OptionalFormatF4, OptionalFormatF8,
          OptionalFormatSize, OptionalFormatVSize, OptionalFormatFSize,
          OptionalFormatEndMarker
      }
  }
  ```

  Refer to the encoding discussion for more information on the meaning of these values.

- `void destroy()`
  Applications must call this function in order to reclaim resources.

Here is a simple example that demonstrates how to extract a boolean and a sequence of strings from a stream:

C#

```
byte[] data = ...
Ice.InputStream inStream =
    Ice.Util.createInputStream(communicator, data);
try {
    bool b = inStream.readBool();
    string[] seq = inStream.readStringSeq();
} finally {
    inStream.destroy();
}
```

See Also

- Basic Data Encoding
- Data Encoding for Classes
- Data Encoding for Exceptions
- Serializable Objects in C-Sharp
- Stream Helper Functions in C-Sharp